**Unit 5:**

A **file** is a collection of information, which can be accessed and used by a computer program. Usually, a file resides on a durable storage. Durable means that the data is persistent, i.e. it can be used by other programs after the program which has created or manipulated it has terminated.   
  
The term file management in the context of computers refers to the manipulation of data in a file or files and documents on a computer. Though everybody has an understanding of the term file, we present a formal definition:   
  
**A file or a computer file** is a chunk of logically related data or information which can be used by computer programs. Usually a file is kept on a permanent storage media, e.g. a hard drive disk. A unique name and path is used by human users or in programs or scripts to access a file for reading and modification purposes.

A file is used because real life applications involve large amounts of data and in such situations the console oriented I/O operation pose two major problems:

* First, it becomes cumbersome and time consuming to handle huge amount of data through terminals.
* Second, when doing I/O using terminal, the entire data is lost when either the program is terminated or computer is turned off. Therefore, it becomes necessary to store data on a permanent storage and read whenever necessary, without destroying the data.

**FILEPATH:**

Your computer drive is organized in a hierarchical structure of files and directories.

* files -- These contain information. Examples include be csv files, or python files.
* directories -- These contain files and directories inside of them

**Absolute file paths:** They are notated by a leading forward slash or drive label.

For example, /home/example\_user/example\_directory or C:/system32/cmd.exe. An absolute file path describes how to access a given file or directory, starting from the root of the file system. A file path is also called a pathname.

**Relative file paths:** They are notated by a lack of a leading forward slash.

For example, example\_directory. A relative file path is interpreted from the perspective your current working directory. If you use a relative file path from the wrong directory, then the path will refer to a different file than you intend, or it will refer to no file at all.

In a sense, whenever you use a relative file path, it is joined with your current directory to create an absolute file path.

That is, if my current working directory is /home/example\_user and I use a relative file path of example\_directory/example\_python\_program, then that is equivalent to using the absolute file path /home/example\_user/example\_directory/example\_file\_program.

In the following example usage of a Unix command-line shell, the current working directory is initially /home/example\_user/example\_directory. There is a program called example\_python\_program, which prints "this is an example python program". At first, the program can be referenced by the relative file path example\_python\_program. After the directory is changed to /home/example\_user, the relative file path to access the program becomes example\_directory/example\_python\_program. Please note that the $ symbolizes a prompt where the user is allowed to type.

$ pwd

/home/example\_user/example\_directory

$ ls

example\_python\_program.py

$ python example\_python\_program.py

this is an example python program

$ example\_directory cd ..

$ pwd

/home/example\_user

$ python example\_python\_program.py

python: can't open file 'example\_python\_program.py': [Errno 2] No such file or directory

$ python example\_directory/example\_python\_program.py

this is an example python program

**Types of files:**

**ASCII Text Files:**

A text file is a stream of characters that can be sequentially processed by a computer in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing or appending) at any given time. Because text files can process characters, they can only read or write data one character at a time.

**Binary Files:**

A binary file is a file which may contain any type of data, encoded in binary form for computer storage and processing purposes. It includes files such as word processing documents, pdfs, images, spreadsheets, videos, zip files and other executable programs. Like a text file, a binary file is a collection of bytes.

**Opening and Closing files:**

A **file object** allows us to use, access and manipulate all the user accessible files. One can read and write any such files.

When a file operation fails for an I/O-related reason, the exception IO Error is raised. This includes situations where the operation is not defined for some reason, like seek() on a tty device or writing a file opened for reading.

**Files have the following methods:**

1. **open():**Opens a file in given access mode.

open(file\_address, access\_mode)

**Examples of accessing a file:** A file can be opened with a built-in function called open(). This function takes in the file’s address and the access\_mode and returns a file object.  
There are different types of access\_modes:

**r:** Opens a file for reading only

**r+:** Opens a file for both reading and writing

**w:** Opens a file for writing only

**w+:** Open a file for writing and reading.

**a:** Opens a file for appending

**a+:** Opens a file for both appending and reading

When you add 'b' to the access modes you can read the file in binary format rather than the default text format. It is used when the file to be accessed is not in text.

1. **close()**: Used to close an open file. A closed file cannot be read or written any more.

|  |
| --- |
| # Opening and closing a file  f = open(\_\_file\_\_, 'r')    #close()  f.close() |
|  |
|  |

**Different File Methods:**

|  |  |
| --- | --- |
| **S.No.** | **Methods with Description** |
| 1 | [**file.close()**](https://www.tutorialspoint.com/python/file_close.htm)  Close the file. A closed file cannot be read or written any more. |
| 2 | [**file.flush()**](https://www.tutorialspoint.com/python/file_flush.htm)  Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects. |
| 3 | [**file.fileno()**](https://www.tutorialspoint.com/python/file_fileno.htm)  Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system. |
| 4 | [**file.isatty()**](https://www.tutorialspoint.com/python/file_isatty.htm)  Returns True if the file is connected to a tty(-like) device, else False. |
| 5 | [**file.next()**](https://www.tutorialspoint.com/python/file_next.htm)  Returns the next line from the file each time it is being called. |
| 6 | [**file.read([size])**](https://www.tutorialspoint.com/python/file_read.htm)  Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). |
| 7 | [**file.readline([size])**](https://www.tutorialspoint.com/python/file_readline.htm)  Reads one entire line from the file. A trailing newline character is kept in the string. |
| 8 | [**file.readlines([sizehint])**](https://www.tutorialspoint.com/python/file_readlines.htm)  Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read. |
| 9 | [**file.seek(offset[, whence])**](https://www.tutorialspoint.com/python/file_seek.htm)  Sets the file's current position |
| 10 | [**file.tell()**](https://www.tutorialspoint.com/python/file_tell.htm)  Returns the file's current position |
| 11 | [**file.truncate([size])**](https://www.tutorialspoint.com/python/file_truncate.htm)  Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size. |
| 12 | [**file.write(str)**](https://www.tutorialspoint.com/python/file_write.htm)  Writes a string to the file. There is no return value. |
| 13 | [**file.writelines(sequence)**](https://www.tutorialspoint.com/python/file_writelines.htm)  Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. |

1. **read([size])**: It reads the entire file and returns it contents in the form of a string. Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes). If the size argument is negative or omitted, read all data until EOF is reached.

|  |
| --- |
| # Reading a file  f = open(\_\_file\_\_, 'r')    #read()  text = f.read(10)    print(text)  f.close() |
|  |

1. **readline([size])**: It reads the first line of the file i.e till a newline character or an EOF in case of a file having a single line and returns a string. If the size argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. An empty string is returned only when EOF is encountered immediately.

|  |
| --- |
| # Reading a line in a file  f = open(\_\_file\_\_, 'r')    #readline()  text = f.readline(20)  print(text)  f.close() |

1. **readlines([sizehint])**: It reads the entire file line by line and updates each line to a list which is returned. Read until EOF using readline() and return a list containing the lines thus read. If the optional size hint argument is present, instead of reading up to EOF, whole lines totalling approximately size hint bytes (possibly after rounding up to an internal buffer size) are read.

|  |
| --- |
| # Reading a file  f = open(\_\_file\_\_, 'r')    #readline()  text = f.readlines(25)  print(text)  f.close() |

1. **write(string)**: It writes the contents of string to the file. It has no return value. Due to buffering, the string may not actually show up in the file until the flush() or close() method is called.

|  |
| --- |
| # Writing a file  f = open(\_\_file\_\_, 'w')  line = 'Welcome Geeks\n'    #write()  f.write(line)  f.close() |

1. **More Examples in different modes:**

|  |
| --- |
| # Reading and Writing a file  f = open(\_\_file\_\_, 'r+')  lines = f.read()  f.write(lines)  f.close() |

# Writing and Reading a file

f = open(\_\_file\_\_, 'w+')

lines = f.read()

f.write(lines)

f.close()

# Appending a file

f = open(\_\_file\_\_, 'a')

lines = 'Welcome Geeks\n'

f.write(lines)

f.close()

# Appending and reading a file

f = open(\_\_file\_\_, 'a+')

lines = f.read()

f.write(lines)

f.close()

1. **writelines(sequence)**: It is a sequence of strings to the file usually a list of strings or any other iterative data type. It has no return value.

|  |
| --- |
| # Writing a file  f = open(\_\_file\_\_, 'a+')  lines = f.readlines()    #writelines()  f.writelines(lines)  f.close() |
|  |

1. **tell()**: It returns an integer that tells us the file object’s position from the beginning of the file in the form of bytes

|  |
| --- |
| # Telling the file object position  f = open(\_\_file\_\_, 'r')  lines = f.read(10)    #tell()  print(f.tell())  f.close() |
|  |
|  |

1. **seek(offset, from\_where)**: It is used to change the file object’s position. Offset indicates the number of bytes to be moved. from\_where indicates from where the bytes are to be moved.

|  |
| --- |
| # Setting the file object position  f = open(\_\_file\_\_, 'r')  lines = f.read(10)  print(lines)    #seek()  print(f.seek(2,2))  lines = f.read(10)  print(lines)  f.close() |

1. **flush()**: Flush the internal buffer, like stdio‘s fflush(). It has no return value. close() automatically flushes the data but if you want to flush the data before closing the file then you can use this method.

|  |
| --- |
| # Clearing the internal buffer before closing the file  f = open(\_\_file\_\_, 'r')  lines = f.read(10)    #flush()  f.flush()  print(f.read())  f.close() |

1. **fileno()**: Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.

|  |
| --- |
| # Getting the integer file descriptor  f = open(\_\_file\_\_, 'r')    #fileno()  print(f.fileno())  f.close() |
|  |

1. **next()**: It is used when a file is used as an iterator. The method is called repeatedly. This method returns the next input line or raises Stop Iteration at EOF when the file is open for reading( behaviour is undefined when opened for writing).

|  |
| --- |
| # Iterates over the file  f = open(\_\_file\_\_, 'r')    #next()  try:      while f.next():          print(f.next())  except:      f.close() |
|  |

1. **isatty()**: Returns True if the file is connected to a tty(-like) device and False if not.

|  |
| --- |
| # Checks if file is connected to a tty(-like) device  f = open(\_\_file\_\_, 'r')    #isatty()  print(f.isatty())  f.close() |
|  |

1. **truncate([size])**: Truncate the file’s size. If the optional size argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed. Note that if a specified size exceeds the file’s current size, the result is platform-dependent: possibilities include that the file may remain unchanged, increase to the specified size as if zero-filled, or increase to the specified size with undefined new content.

|  |
| --- |
| # Truncates the file  f = open(\_\_file\_\_, 'w')    #truncate()  f.truncate(10)  f.close() |

**File object attributes:**

* **closed**: returns a Boolean indicating the current state of the file object. It returns true if the file is closed and false when the file is open.
* **encoding**: The encoding that this file uses. When Unicode strings are written to a file, they will be converted to byte strings using this encoding.
* **mode**: The I/O mode for the file. If the file was created using the open() built-in function, this will be the value of the mode parameter.
* **name**: If the file object was created using open(), the name of the file.
* **newlines**: A file object that has been opened in universal newline mode have this attribute which reflects the newline convention used in the file. The value for this attribute are “\r”, “\n”, “\r\n”, None or a tuple containing all the newline types seen.
* **softspace**: It is a Boolean that indicates whether a space character needs to be printed before another value when using the print statement.

|  |
| --- |
| f = open(\_\_file\_\_, 'a+')  print(f.closed)  print(f.encoding)  print(f.mode)  print(f.newlines)  print(f.softspace) |
|  |

**Opening files using with keyword:**

With the “with” statement, you get better syntax and exceptions handling.

You can also work with file objects using the with statement. It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it’s good practice to use the with statement where applicable.

One bonus of using this method is that any files opened will be closed automatically after you are done. This leaves less to worry about during clean-up.

To use the with statement to open a file:

with open(“filename”) as file:

Few examples.

with open(“testfile.txt”) as file:

data = file.read()

do something with data

 You can also call upon other methods while using this statement. For instance, you can do something like loop over a file object:

with open(“testfile.txt”) as f:

for line in f:

print line,

You’ll also notice that in the above example we didn’t use the “file.close()” method because the with statement will automatically call that for us upon execution. It really makes things a lot easier, doesn’t it?

To better understand the with statement, let’s take a look at other examples:

To write to a file using the with statement:

with open(“hello.txt”, “w”) as f:

f.write(“Hello World”)

To read a file line by line, output into a list:

with open(“hello.txt”) as f:

data = f.readlines()

This will take all of the text or content from the “hello.txt” file and store it into a string called “data”.

**Splitting Lines in a Text File**

There is a unique function that allows you to split the lines taken from a text file. What this is designed to do, is split the string contained in variable data whenever the interpreter encounters a space character.

But just because we are going to use it to split lines after a space character, doesn’t mean that’s the only way. You can actually split your text using any character you wish - such as a colon, for instance.

The code to do this (also using a with statement) is:

with open(“hello.text”, “r”) as f:

data = f.readlines()

for line in data:

words = line.split()

print words

If you wanted to use a colon instead of a space to split your text, you would simply change line.split() to line.split(“:”).

The output for this will be:

[“hello”, “world”, “how”, “are”, “you”, “today?”]

[“today”, “is”, “Saturday”]

The reason the words are presented in this manner is because they are stored – and returned – as an array. Be sure to remember this when working with the split function.

**Renaming and Deleting files:**

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

# **The rename () Method:** The rename() method takes two arguments, the current filename and the new filename.

#### Syntax

os.rename(current\_file\_name, new\_file\_name)

#### Example

Following is the example to rename an existing file test1.txt:

#!/usr/bin/python

import os

# Rename a file from test1.txt to test2.txt

os.rename( “test1.txt”, “test2.txt” )

# **The remove ( ) Method:** You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.

  Syntax

os.remove(file\_name)

#### Example: Following is the example to delete an existing file test2.txt:

#!/usr/bin/python

import os

# Delete file test2.txt

os.remove(“text2.txt”)

# **Directories in Python:** All files are contained within various directories, and Python has no problem handling these too. The os module has several methods that help you create, remove, and change directories.

# The mkdir() Method: You can use the mkdir() method of the os module to create directories in the current directory.

You need to supply an argument to this method which contains the name of the directory to be created.

#### Syntax

os.mkdir(“newdir”)

Example: Following is the example to create a directory test in the current directory:

#!/usr/bin/python

import os

# Create a directory

os.mkdir(“test”)

# The chdir() Method: You can use the chdir() method to change the current directory. The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

#### Syntax

  os.chdir(“newdir”)

Example: Following is the example to go into “/home/newdir” directory:

#!/usr/bin/python

import os

# Changing a directory to “/home/newdir”

os.chdir(“/home/newdir”)

# The getcwd() Method: The getcwd() method displays the current working directory.

  Syntax

os.getcwd()

#### Example: Following is the example to give current directory:

#!/usr/bin/python

import os

# This would give location of the current

os.getcwd()

# The rmdir() Method: The rmdir() method deletes the directory, which is passed as an argument in the method. Before removing a directory, all the contents in it should be removed.

Syntax

os.rmdir(‘dirname’)

#### Example: Following is the example to remove “/tmp/test” directory. It is required to give fully qualified name of the directory, otherwise it would search for that directory in the current directory.

#!/usr/bin/python

import os

# This would remove “/tmp/test” directory.

os.rmdir( “/tmp/test” )

**Methods from the os module:**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Method Name** | **Description** |
| 1 | os.path.abspath() | To convert relative path to absolute path |
| 2 | os.path.isabs(path) | Accepts a file path as an argument and returns True if the path is an absolute path and false otherwise. |
| 3 | os.path.relpath(path, start) | Accepts a file path and a start as an argument and returns relative path that begins from start. If start is not given, the current directory is taken as start. |
| 4 | os.path.dirname(path) | Returns a string that includes everything specified in the path that comes before the last slash. |
| 5 | os.path.basename(path) | Returns a string that includes everything specified in the path that comes after the last slash. |
| 6 | os.path.split(path) | Accepts a file path and returns its directory name as well as the basename. |
| 7 | os.path.getsize(path) | Returns the size of the file specified in the path argument |
| 8 | os.listdir(path) | Returns a list of filenames in the specified path. |
| 9 | os.path.exists(path) | Accepts a path as an argument and returns True if the file or folder specified in the path exists and False otherwise. |
| 10 | os.path.isfile(path) | Accepts a path as an argument and returns True if the path specifies a file and False otherwise. |
| 11 | os.path.isdir(path) | Accepts a path as an argument and returns True if the path specifies an existing directory and False otherwise. |